Evolving Object-Oriented Designs with Refactorings

Lance Tokuda and Don Batory
Department of Computer Science
University of Texas at Austin
{unicron, batory}@cs.utexas.edu

Abstract[1]

Refactorings are behavior-preserving *program transformations that automate design level changes in object-oriented applications. Our previous research established that many schema transformations, design patterns, and hotspot meta-patterns are automatable. This research evaluates whether refactoring technology can be transferred to the mainstream by restructuring non-trivial C++ applications. The applications that we examine were evolved manually by software engineers. We show that an equivalent evolution could be reproduced significantly faster and cheaper by applying a handful of general-purpose refactorings. In one application, over 14K lines of code were transformed automatically that otherwise would have been coded by hand. Our experiments identify benefits, limitations, and topics of further research related to the transfer of refactoring technology to a production environment.*

1. Introduction

Before the invention of *graphical user interface (GUI)* editors, the process of evolving a GUI was to design, code, test, evaluate, and redesign again. With the introduction of editors, GUI design has become an interactive process allowing users to design, evaluate, and redesign an interface on-screen and to output compilable source code that reflects the latest design.

We believe that a similar advance needs to occur for editing object-oriented class diagrams. Editing a class diagram can be as simple as adding a line between classes to represent an inheritance relationship or moving a variable from a subclass to a superclass. However, such changes must now be accompanied by painstakingly identifying lines of affected source code, manually updating the source, testing the changes, fixing bugs, and retesting the application until the risk of new errors is sufficiently low.

Just as GUI editors revolutionized GUI design, we believe that class diagram editors (where changes to an application's diagram automatically trigger corresponding changes to its underlying source code) will revolutionize the evolution of software design. The technology to power such a tool is *refactorings* — behavior-preserving program transformations that automate many design level[2] changes. The term *automate* refers to a refactoring's programmed check for enabling conditions and its execution of all source code changes. The choice of which design to implement and which refactorings need to be applied is always made by a human. Refactorings preserve behavior thereby reducing costly and tedious debugging and testing that would otherwise have to be performed.

2. Refactorings

A *refactoring* is a parameterized behavior-preserving program transformation that updates an application's design and underlying source code. A refactoring is typically a simple transformation that has a straightforward (but not necessarily trivial) impact on application source. An example is **inherit[Base, Derived]**, which establishes a superclass-subclass relationship between two classes, **Base** and **Derived**, that were previously unrelated. From the perspective of an object-oriented class diagram, the **inherit** refactoring merely adds an inheritance relationship between the **Base** and **Derived** classes, but also it alters the application's source code to reflect this change.

A summary of the class diagram notation used throughout this paper is presented in Figure 2.1.

2.1. Enabling Conditions

Programs are restructured by applying a series of refactorings. Because individual refactorings preserve behavior, a series of refactorings also preserves behavior. To preserve behavior, we adopt the method proposed by Banerjee and Kim for database schema evolutions [2] and employed by

1. We gratefully acknowledge the sponsorship of the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226) and the University of Texas at Austin Applied Research Laboratories.

2. We use a limited definition of the term *design* referring to the aspect of design reflected in the extended class diagram notation from Gamma [1].

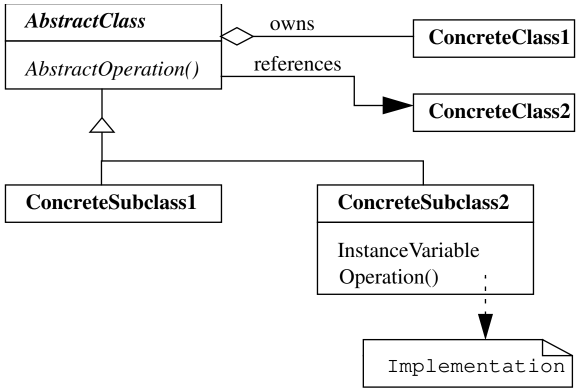


Figure 2.1: Notation

1. ***Base*** must not be a subclass of **Derived** and **Derived** must not have a superclass.
2. Member variables of **Derived** must have distinct names from member variables of ***Base*** and its superclasses.
3. A member function of **Derived** which overrides a function must have the same type signature as the function it overrides.
4. Subclasses of ***Base*** must implement any pure virtual methods if objects of that class are created.
5. Initializer lists must not be used to initialize **Derived** objects.
6. For all inherited instance variables whose type is a class, the constructors for those classes cannot have any side-effects outside of object initialization if **Derived** is instantiated.
7. Program behavior must not depend on the size or layout of **Derived**.

Figure 2.2: Inherit[*Base*, Derived] enabling conditions

Opdyke for refactorings [3]. Opdyke claimed to identify a set of seven invariants which, if preserved, guaranteed that two Smalltalk or C++ programs would run identically. For example, his first invariant is that each class must have a unique superclass and its superclass must not also be one of its subclasses. When a refactoring runs the risk of violating an invariant, enabling conditions are added to guarantee that the invariant is preserved. Enabling conditions for the **inherit** refactoring are listed in Figure 2.2. **Inherit**'s first enabling condition preserves Opdyke's first invariant.

2.2. Design Evolution and Refactorings

Three kinds of object-oriented design evolution are: schema transformations, design pattern microarchitectures, and hot-spots. *Schema transformations* are drawn from object-oriented database schema transformations that perform edits on a class diagram [2]. Examples are adding new instance variables and moving methods up the class hierarchy. *Design patterns* are recurring sets of relationships between classes, objects, methods, etc. that define preferred solutions to common object-oriented design problems [1]. *Hot-spots* are aspects of a program which are likely to change from application to application [4]. Designs using abstract classes and template methods are prescribed to keep these hot-spots flexible.

In previous work, we identified a list of schema transformations, design pattern microarchitectures, and hot-spot meta patterns that were automatable with refactorings [5]. Refactorings to enable these changes includes those proposed by Banerjee and Kim for evolving object-oriented database schemas [2] and by Opdyke for restructuring object-oriented programs [3]. We found that transforming actual C++ programs required additional refactorings. We enlarged the set of schema evolutions to include, for example, **inherit** (from the example above) and **substitute**. **Substitute** changes a class's dependency on a class **C1** to a dependency on a superclass of **C1** [6]. Other refactorings are language-specific; **procedure_to_method** and **structure_to_class** convert C artifacts to their C++ equivalents. Yet another set of refactorings supports the addition of design pattern microarchitectures in evolving programs [5, 6]. Examples include **add_factory_method**, **singleton**, and **procedure_to_command**. **Add_factory_method** creates a method which returns a new object, **singleton** ensures that a class will have only one instance, and **procedure_to_command** converts a C procedure to a singleton class with a method for executing the procedure. The refactorings that we added to the lists of Banerjee, Kim, and Opdyke are italicized in Table 1.

<u>Schema Refactorings</u>	pull_up_method
add_variable	move_method_across_ object_boundary
create_variable_accessor	extract_code_as_method
create_method_accessor	*declare_abstract_method*
rename_variable	*structure_to_pointer*
remove_variable	
push_down_variable	<u>C++ Refactorings</u>
pull_up_variable	*procedure_to_method*
move_variable_across_ object_boundary	*structure_to_class*
create_class	
rename_class	<u>Pattern Refactorings</u>
remove_class	*add_factory_method*
inherit	*create_iterator*
uninherit	*composite*
substitute	*decorator*
rename_method	*procedure_to_command*
remove_method	*procedure_ptr_to_command*
push_down_method	*singleton*

Table 1: Object-oriented refactorings

The first implementations of Opdyke's refactorings were by Tokuda for C++ [6] and Roberts for Smalltalk [7]. To our knowledge, we were first to implement refactorings for design patterns [6] and hot-spot meta patterns [5].

Given that schema transformations, design patterns, and hot-spot meta patterns are used frequently in evolving designs, many of which are automatable as (one or more) refactorings, we expected to replicate some, but not all, design changes in our experiments.

3. Evolving Applications

We selected SEMATECH's CIM Works and CMU's Andrew User Interface System as examples of evolving applications. They were chosen based on availability of source code with a version history, size, and presence of design changes. The following features make our study unique:

Replication of design evolution. Designs were extracted from two versions of the same application. The older design became the initial state and the newer design became the target state. Our objective was to determine if a sequence of refactorings could be applied to transform the initial state to the target state. By doing so, we would automate changes that were performed manually by the original application designers. This correspondence makes comparison of automation versus hand-coding valid and provides us with a key indicator: how often refactorings could be used.

Non-trivial Applications. Transforming large applications tests refactoring scalability. Ideas that are effective on small applications of fewer than one thousand lines of code may ultimately fail for real world applications whose size can exceed one hundred thousand lines.

Mainstream object-oriented language. C++ was chosen as the target language for experimentation. It is by far the most widespread object-oriented programming language for practical reasons such as backward compatibility with C, portability, availability of third party compilers and tools, legacy system compatibility, and availability of trained personnel. It was expected that C++'s complexity might introduce problems which would not appear for less popular object-oriented languages. A side benefit of this choice is that most claims for C++ can also be made for the increasingly popular Java programming language.

3.1. Evolving CIM Works

Computer Integrated Manufacturing (CIM) Framework is an industry-wide initiative to define a standardized object-oriented framework for writing semiconductor manufacturing execution systems [8]. CIM Works is a Windows application created to demonstrate and test the SEMATECH CIM Framework specification [9].

Major design changes in CIM Works occur between Version 2 and Version 4. The Version 2 design shown in Figure 3.1 stores data and its graphical representation in the same object. For example, **CEquipmentManager** contains methods for adding and removing pieces of equipment to be managed as well as methods for building a GUI menu. The Version 4 design shown in Figure 3.2 separates data and graphics into two class hierarchies. This separation gave Version 4 the freedom to create different views of the same data as with the model-view-controller paradigm [10].

Version 2 is approximately 11K lines of code. The transformation between designs is accomplished in nine steps, each of which is realized by applying a sequence of primitive refactorings:

1. Rename the classes of the original hierarchy to the split hierarchy using **rename_class**. (The original classes retain the GUI aspects of objects, whereas their corresponding "split" classes — created in Steps 2 or 7 — encapsulates object data).
2. Create the concrete data classes **Factory**, **Person**, **EquipManager**, etc. using **create_class**.
3. Add `m_objptr` instance variables to the concrete GUI classes using **add_variable**. `m_objptr` is of the corresponding data class type (Figure 3.3a).
4. Create abstract data classes **Resource**, **CompManager**, **MovementResource**, etc. using **create_class.**
5. Establish inheritance relationships between the abstract data classes and the concrete data classes using **inherit** (Figure 3.3b).
6. Move non-GUI instance variables and methods from the GUI classes to the data classes using **move_variable_across_object_boundary** and **move_method_across_object_boundary**. Data is accessed through the `m_objptr` instance variables (Figure 3.4).
7. Move common instance variables and method declarations up the data class hierarchy using **pull_up_variable** and **declare_virtual_method** (Figure 3.5).
8. Change the type of `m_objptr` from a structure to a pointer using **structure_to_pointer**.
9. Declare the reference between GUI objects and data objects in the abstract classes. References to data objects are made abstract (Figure 3.2).[3]

These steps were executed by 81 refactorings, resulting

3. In this step, the generalization is made that all **CIcon** objects point to a **Resource** object through the `m_objptr` instance variable. This requires that casts to the appropriate data class are made whenever data object instance variables are referenced through GUI objects. For example:

```
CIcPerson *p = new CIcPerson;
p->person_ptr->f_name = "John";
```

is transformed to:

```
CIcPerson p;
((Person *)p->m_objptr)->f_name =
    "John";
```

It is unclear if this was the correct design decision since the GUI classes are specific to a single data class. This step was not automated although it would be possible to do so.

Figure 3.1: Version 2 Design

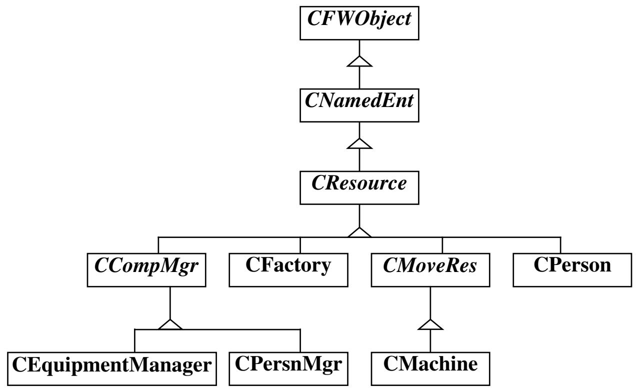


Figure 3.2: Version 4 Design

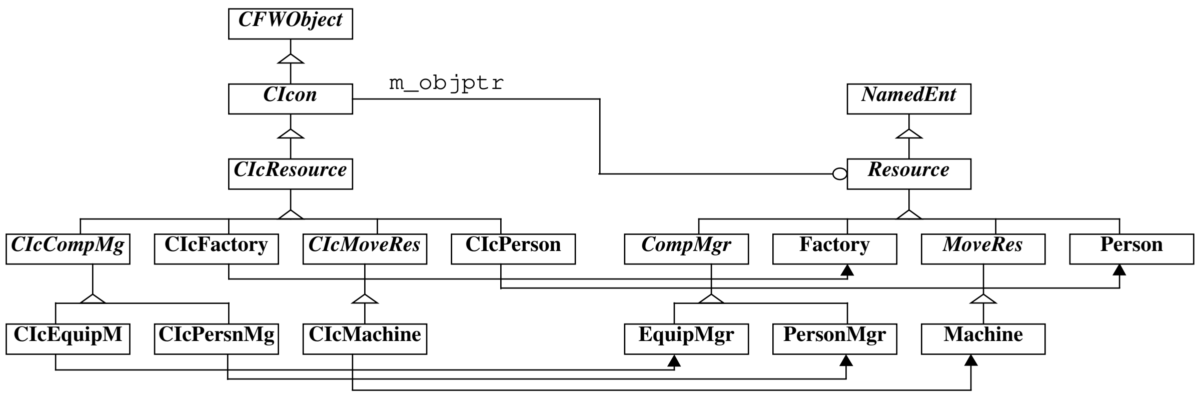


Figure 3.3a: Connect GUI and Data Classes

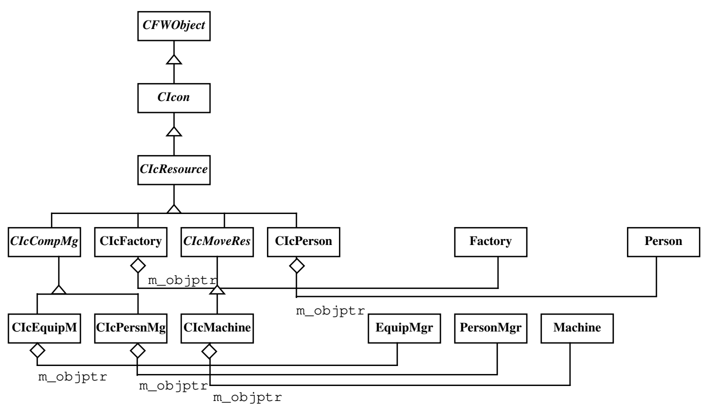
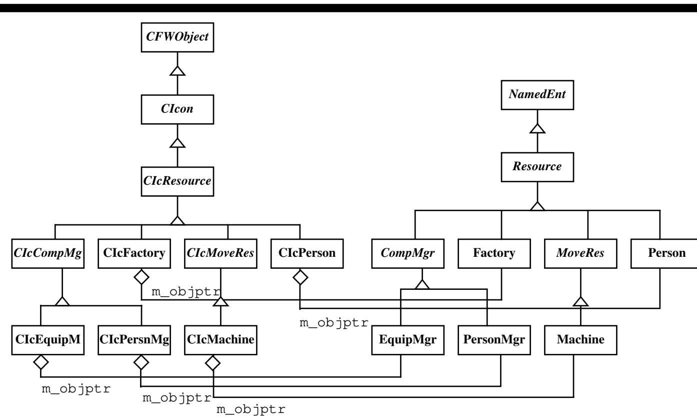


Figure 3.3b: Create Abstract Data Class Hierarchy



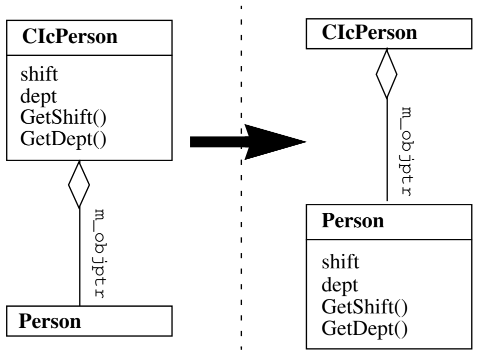


Figure 3.4: Instance variables and methods moved to data classes

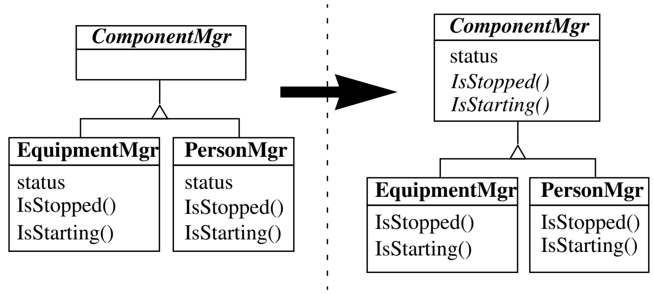


Figure 3.5: Instance variables and method declarations moved to abstract classes

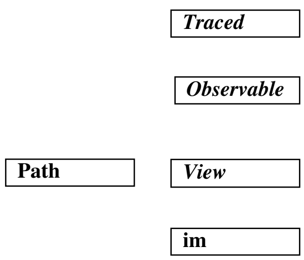


Figure 3.6: Structures converted to classes

in a total of 486 lines of CIM Works source being modified.

3.2. Evolving the Andrew User Interface System

The *Andrew User Interface System (AUIS)* from CMU is an integrated set of tools that allow users to create, use, and mail documents and applications containing typographically-formatted text and embedded objects [11]. The two versions under study were Version 6.3 written in C and Version 8.0 converted to C++. Version 6.3 stores actions as function pointers while Version 8.0 supports and recommends creation of a separate subclass for each action (similar to the Command design pattern[4]). Over ninety classes using almost 800 actions are affected. The transformation is accomplished in five steps.

1. Convert Version 6.3 C structures to C++ classes using **structure_to_class** (Figure 3.6).
2. Create the **ATK** and **Command** abstract classes using **create_class**.
3. Establish the inheritance relationships between **ATK** and other classes using **inherit**.
4. Derive command classes for each action using **procedure_to_command**. Figure 3.8 displays the result of transforming `PlayKbdMacro()` into a **Command** subclass. The newly created **PlayKbdMacroCmd** contains an `Execute()` method which calls `PlayKbd-Macro()`. It also contains an `Instance()` method which returns a unique instance of the class. Using `Instance()` instead of `new` to create objects guarantees that a pointer to a **PlayKbdMacroCmd** object is unique.
5. Convert procedure pointers to commands using **procedure_ptr_to_command**. In this step, the data types for structures using procedure pointers are converted to use **Command** pointers, procedure calls are converted to use `Execute()` methods, and procedure assignments are converted to use `Instance()` methods. Figure 3.9 displays the transformation of the **bind_Description** structure. The `proc` instance variable

4. The Command design pattern objectifies an action. The action is triggered by calling an `Execute()` method implemented in each derived class [1].

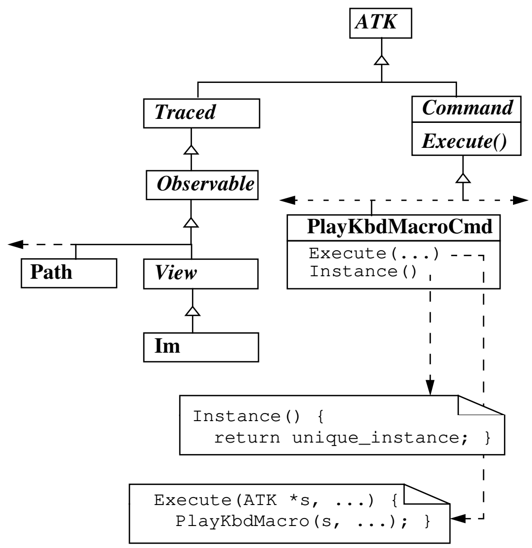


Figure 3.8: Software Microarchitecture for Im and Command classes

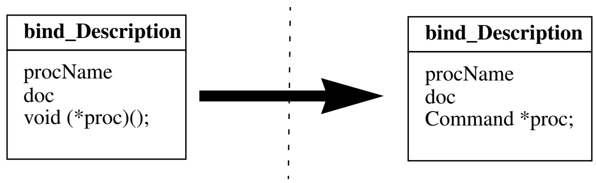


Figure 3.9: Convert procedure pointer to Command pointer

is converted to a **Command** pointer.

All steps were executed with approximately 800 refactorings resulting in 14K lines of code changes[5].

4. Introspection and Lessons Learned

Our experiments provided a tremendous learning experience in evaluating refactoring technologies. In the following sections, we present some of the more important lessons that we learned on refactoring benefits, limitations, and research problems that must be solved for refactoring technology to succeed. Further discussion is given in [12].

4.1. Refactoring Benefits

Automating Design Changes. The most important result of our research is to establish that refactorings can automate significant design changes involving thousands of lines of code in real world applications. It is of interest to compare the effort required to perform these changes manually versus the effort when aided by refactorings. *We estimate that the CIM Works changes would take us two days to implement and debug by hand versus two hours when aided by refactorings. We estimate that the AUIS changes would require two weeks to implement and debug by hand versus one day when aided by refactorings.*

Reduced Testing. A good refactoring implementation can reduce the effort required to test new designs. When refactorings preserve behavior, only hand coded changes need to be tested.

Simpler Designs. Refactorings reduce the need for overly complex designs. Gamma et. al. note that a common design pattern pitfall is over-enthusiasm: “Patterns have costs (indirection, complexity) therefore [one should] design to be as flexible as needed, not as flexible as possible”. Designs which attempt to anticipate too many future extensions may also be more error prone with less static type checking[6]. Refactorings are capable of extending designs in multiple ways. They encourage designers to create lean designs for the task at hand and to extend those designs with refactorings as new capabilities are needed.

Validation Assistance. Enabling condition checks can detect conflicts between a code level implementation and a desired design change. For example, a programmer may decide to move an instance variable from a base class to a derived class without realizing that objects of the base class access the instance variable being moved. Enabling condition checks will detect this error. Refactorings are capable of detecting errors resulting from a long series of changes which would be costly to perform and undo manually.

Ease of Exploration. Refactorings allow designers to experiment with new designs. While schema evolutions and design patterns are manually coded into applications today, it is clear that automating their introduction will allow designers to more easily explore a design space without major commitments in coding and debugging time.

4.2. Refactoring Limitations

Experiments with large applications revealed limitations which were not issues in previous work on small proof-of-concept programs. We discuss our most important observations to alert future researchers to the problems that they will face.

Preprocessor Directives. Our C++ program transformation tool cannot deal with preprocessor directives because preprocessor directives are not part of the C++ language. The programs in our experiments were preprocessed before being transformed and at that point, preprocessor information could no longer be recovered. While we believe that workarounds are possible for the majority of the cases, it is generally not possible to handle all problems that arise in large software applications.

Conservative Enabling Conditions. Refactorings have been found to be useful even when predicated on conservative enabling conditions. For example, the **inherit** transformation is conservatively limited to single inheritance systems by Opdyke’s first invariant. While support for multiple inheritance systems is possible, it was not necessary for transforming the applications described in this paper or for adding numerous design patterns and hot-spot meta patterns [5].

Automated Verification of Enabling Conditions. Most but not all enabling conditions can be verified automatically. For the **inherit** refactoring, the first five conditions can be checked automatically (Figure 2.2).

Opdyke identifies a condition which cannot be verified automatically: program behavior must not be dependent on

5. This number is large because AUIS used 800 actions implemented as procedures and the conversion of a procedure to a command required a transformation. More refactorings did not imply more complexity. We found it was easier to choose the refactorings for AUIS than for CIM Works because conceptually, the evolution of AUIS’s design required only five steps.

6. Many design patterns use runtime composition versus inheritance as an extension mechanism [1]. The dynamic nature of composition precludes static typechecking.

the size or layout of objects [3]. Size and layout were not issues with the two programs transformed in this paper or other programs transformed in [6, 7], however, users of refactorings must be aware of this limitation.

Behavior Preservation. Our experiments revealed that preservation of Opdyke's invariants was not sufficient to guarantee preservation of behavior. Three additional invariants required for C++ are:

1. **Implementation of pure-virtual functions**. If a class is instantiated, then it cannot have any pure-virtual functions.
2. **Maintaining aggregate objects**. If a program depends on the aggregate property of an object, then that property must be preserved[7].
3. **No instantiation side-effects**. If a refactoring can change the frequency or order in which classes are instantiated, then the constructor cannot have any side-effects beyond initializing the object created.

In light of this discovery, we recognize that refactorings are behavior-preserving due to good engineering and not because of any mathematical guarantee. It is the responsibility of the refactoring designer to identify all enabling conditions necessary to ensure that behavior is preserved. This important issue is explored further in [12].

4.3. Future Research

Our work focused on the practicality of applying primitive refactorings to evolving object-oriented applications. Beyond implementation of required functionality, we identify three issues which require further research.

Granularity of Transformations. The refactorings developed for this research were intended to be primitive and composable to perform more complex refactorings. We did not attempt to minimize the number of refactorings required. In the CIM Works example, the number of refactorings was large (81) although the conceptual number of transformation steps was small (8). One way to reduce the number of refactorings would be to provide larger grain refactorings. In the CIM Works example, the number of refactorings would be significantly reduced if refactorings to move multiple variables and methods were available. Similarly for the Andrew example, most of the 800 transformations take place in Step 4 — converting action procedures to **Command** subclasses. A larger grain transformation which converted a list of procedures to **Command**'s could execute Step 4 in a single transformation. This would reduce the total number of transformations to fewer than twenty. It is important to note that the near term goal of our research has been to develop a basis set of primitive refactorings. Larger grain refactorings up to the size of design patterns may be more convenient in practice.

Program Families. Transformation systems must recognize that many files may be included by multiple programs. When transforming a file used by more than one program, it is desirable for the transformation system to check enabling conditions for all programs in which use that file. Otherwise, a file might be transformed safely for one program while causing another program which uses the same file to break. The situation is further complicated for C++ by conditional compilation flags which imply that different preprocessed versions of a single file should be considered when checking if a transformation can be performed safely.

Integration with Other Tools. Refactorings packaged as individual executables are not dependent on the presence of other tools. In this form, they can be integrated into most mainstream development environments because most environments support command-line access to source code.

Higher levels of integration are still possible. We envision integration with an object-oriented modeling tool such as Rational Rose™ which would allow many refactorings to be invoked as operations on a UML diagram. Integration with a source code control system could allow appropriate files to be checked out, transformed, and checked back in with comments describing the refactorings. Attempts to transform protected files would block the refactoring and notify the user. Integration with an IDE such as Microsoft Visual C++™ would allow transformed code to be displayed immediately in open windows.

4.4. Implications for Java

Java inherits all of C++'s refactoring benefits while avoiding many of its limitations. First, it has no preprocessor which removes a major barrier to a successful C++ implementation. Second, it does not use makefiles which simplifies the process of piecing together the source files to be transformed. Third, code placement is simplified since methods are stored in a file belonging to the class. Java has no free-floating procedures as with hybrid object-oriented languages such as C++. For these reasons coupled with its growing popularity as an internet language, we believe that Java is the best vehicle for transferring refactoring technology to the mainstream.[8] Tools are now being developed to aid in this process [13, 14].

5. Related Work

Bergstein defined a small set of object-preserving class transformations which can be applied to class diagrams

7. **Inherit** destroys this property since aggregates cannot have superclasses.

8. When we began our work, tool support and availability of large Java files were nonexistent. This is no longer true today.

[15]. Lieberherr implemented these transformations in the Demeter object-oriented software environment [16]. Example transformations are deleting useless subclasses and moving instance variables between a superclass and a subclass.

Opdyke coined the term *refactoring* to describe a behavior-preserving program transformation for restructuring object-oriented software systems. Refactorings were inspired by the schema evolutions of Banerjee and Kim [2], the design principles of Johnson and Foote [17] and the design history of the UIUC Choices operating system [18]. An example application of refactorings is the creation of an abstract superclass [3]. Refactorings are implemented for C++ [5, 6, 19] and for Smalltalk [7]. Roberts offers Smalltalk-specific design criteria for a program transformation tool [7]. One criteria which also applies to C++ software is that users should be allowed to name new entities introduced through transformations.

Refactorings are shown to automate the addition of design patterns to object-oriented software systems [5, 6, 7, 19]. Refactorings also support the addition of Pree's [4] hot-spot meta patterns [5].

6. Conclusion

Design evolution is an inevitable manual process often requiring great effort and expense. Refactorings are behavior-preserving program transformations that provide a powerful technology by which significant parts of an application's design evolution can be automated.

The ultimate goal of our research is to provide a mainstream tool that makes editing class diagrams as easy as editing user interfaces with a GUI editor. This paper has taken three important steps towards this goal:

- First, we implemented a set of refactorings that can automate a suite of schema transformations, design patterns, and hot-spot meta patterns. They can reduce or eliminate the need to identify lines of affected source, to execute changes manually, and to test those changes.
- Second, we showed that refactorings can scale and be useful on large, real-world applications. We were able to automate thousands of lines of changes with a general-purpose set of refactorings.
- Third, while our experiments clearly showed the benefits that could result from a refactoring tool, they also revealed the limitations and research problems that remain to be addressed before refactoring technology can be transitioned beyond academic prototypes.

Given the success of our experiments and the difficulty in managing C++ preprocessor information, Java should be the next target language, as we believe that it holds the greatest promise for transferring refactoring technology to the mainstream.

References

[1] E. Gamma et.al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[2] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD*, 1987.

[3] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[4] W. Pree. Meta Patterns — A Means for Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings, ECOOP '94*, Springer-Verlag, 1994.

[5] L. Tokuda and D. Batory. Automating Three Modes of Object-Oriented Software Evolution. In *Proceedings, COOTS '99*, 1999.

[6] L. Tokuda and D. Batory. Automated Software Evolution via Design Pattern Transformations. In *Proc. 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[7] D. Roberts, J. Brant, R. Johnson. A Refactoring Tool for Smalltalk. In *Theory and Practice of Object Systems*, Vol. 3 Number 4, 1997.

[8] S. Stewart. *Roadmap for the Computer-Integrated Manufacturing Application Framework*. NISTIR 5697, June, 1995.

[9] P. McGuire. Lessons learned in the C++ reference development of the SEMATECH computer-integrated manufacturing (CIM) applications framework. In *SPIE Proceedings*, Volume 2913, pages 326-344, 1997.

[10] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. In *Journal of Object-Oriented Programming*, pages 26-49, August 1988.

[11] J. H. Morris et.al. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 1986.

[12] L. Tokuda. *Design Evolution with Refactorings*. Ph.D. thesis, University of Texas, 1999.

[13] I. Baxter. and C. Pidgeon. Software Change Through Design Maintenance. In *Proceedings of the International Conference on Software Maintenance '97*, IEEE Press, 1997.

[14] D. Batory et. al. JTS: Tools for Implementing Domain-Specific Languages. In *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[15] P. Berstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[16] K. Lieberherr, W. Hursch, and C. Xiao. *Object-extending class transformations.* Technical report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston, Massachusetts, 1991.

[17] R. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, June/July 1988.

[18] P. Maydany et.al. A Class Hierarchy for Building Stream-Oriented File Systems. In *Proceedings of ECOOP '89*, Nottingham, UK, July 1989.

[19] W. Scherlis. Systematic Change of Data Representation: Program Manipulations and Case Study. In *Proceedings of ESOP '98*, 1998.